Mandelbrot Set

Name: Xizhen Huang

Matrikel-Nr.: 4889117

Study Program: Distributed Systems Engineering

Email: Xizhen.Huang@mailbox.tu-dresden.de

Table of Contents

1. Description

1.1 Description of the problem

This problem is about calculating Mandelbrot Set. The Mandelbrot set is the set of complex numbers c for which function $f(z) = z^2 + c$ does not diverge when iterate from z=0. In this task, the input is 230, 790 and 24000, that means the matrix size is 230*790, 230 rows and 790 columns, and for each point, number of iterations is 24000. In this task, there are 230*790 = 181700 points need to be calculated and judged if belong to Mandelbrot Set. According to the requirement document, original code needs to be covered into parallel code and works on 56 cores machine correctly.

1.2 Description of unmodified algorithm

The unmodified algorithm is using straightforward way to calculation. In this code, a two-dimensional array is created to store results, and iterating row by row from 0 to 229, and calculating 790 times for each row. If a point meet Mandelbrot Set conditions, corresponding position in result array would be stored '#', otherwise '.'. In unmodified code, only one main thread runs in one CPU, therefore, only one value can be calculated at a time, the calculation process needs to be repeated 230*790 times to finish the whole graph.

1.3 Complexity analysis of the existing algorithm

Time complexity is O(max_row*max_colomn*iteration). Almost, given r rows, c columns and i iterations then the running time is O(r*c*i). But with this extra condition it's not clear how many times will the inner while loop run in total. it will be less than r*c*i times, so O(r*c*i) is still the upper bound.

Space complexity is *O(max_row*max_colomn)*. There is a two-dimensional array is allocated in the unmodified code, size is max_row*max_colomn (230*790), and space of other variables is static, not change with the data scale.

2. Optimization Strategies

According to analysis properties of Mandelbrot Set, it is clear that there is no any dependence between any two points. So, the calculation work could be divided into different parts and assigned to different threads, these threads could do their job concurrently without influencing others. Following two strategies were used in my code.

2.1 Load-balancing

The main goal of load-balancing is assigning almost same workload to each thread, so that some threads would not be bottleneck of the whole program because of overload. In my code, splitting each row of the matrix into a task, so there are 230 tasks. Assuming there are N threads could be used now, if N is equal or larger than task numbers, each thread may be assigned 0 or 1 tasks, but if N is smaller than task numbers each thread may be assigned more than 1 task.

The following figure shows the situation which there are 56 threads (from 0 to 55) are created to do this task.

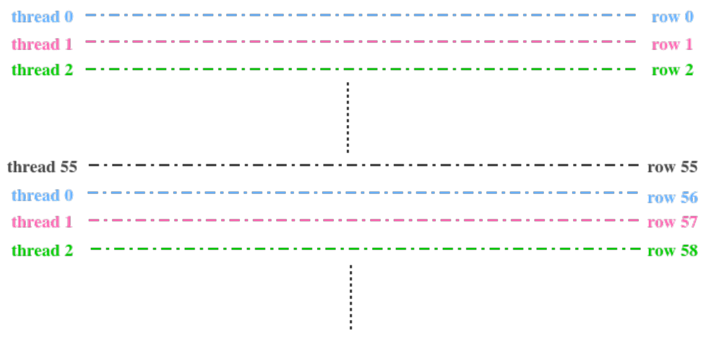

Figure 1. load-balancing working pattern

According to the basic properties of Mandelbrot Set, it can be known that it takes different time to calculate different parts. If point *c* belongs to the set, program needs more time to judge it than those points not belong to the set. Assign tasks to threads in order to ensure that each thread can both have compute-intensive tasks and easy tasks, avoiding overload.

2.2 Static Partitioning

In this strategy, the matrix is split into N blocks for N threads, so there are N tasks, and each thread are only assigned one task (one block). Using **max_row/N** to get the length of the block, and every block has same width is **max_column**.

The following figure shows the situation which there are 56 threads (from 0 to 55) are created to do this task.

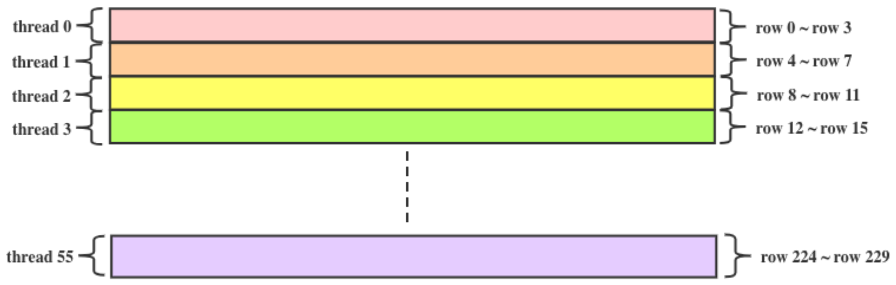

Figure 2. static partitioning working pattern

It can be seen that each thread needs to compute one block. All blocks have same width and length except the latest block. But workload of different threads is different in this strategy. If a thread's block contains more points belong to Mandelbrot Set, it takes more time for this thread to finish its task than others, this thread would be 'bottleneck' of this program. This is why speedup of this algorithm is not that good.

3. Evaluation of parallelization

3.1 Evaluation of Load-balancing

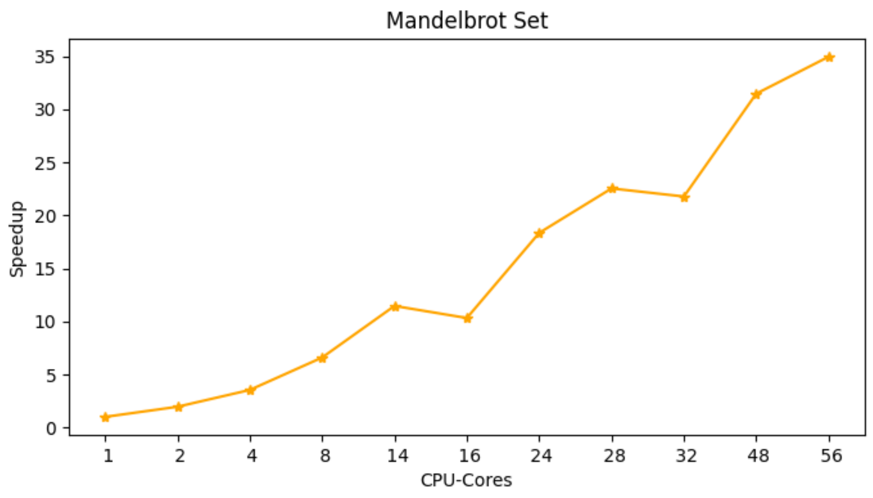

Figure 3. speedup x35 of optimization algorithm

Multithreads is used in this algorithm to achieve parallel execution on 56 cores machine. The figure above shows how the speedup changes with the number of cores by using **Load-Balancing** strategy. It can be concluded from the above graph that this strategy can achieve a speedup up to 35 while using 56 CPU cores. With the increase in the number of CPU cores, the initial acceleration ratio close to linear growth.

Accroding to Amdahl's Law,

$$S_{latency}(s) = \frac{1}{1-p+\frac{p}{s}}$$

where is the theoretical speedup of the execution of the whole task, is the speedup of the part of the task that benefits from improved system resources and is the proportion of execution time that the part benefiting from improved resources originally occupied. It can be calculated from the evaluation of the strategy that in the promising strategy, the parallelization part accounts for 99.09% of the whole execution time in 28 cores, while it accounts for 98.9% of the whole execution time in 56 cores. The overhead of serialization parts increases as the number of threads grows.

In this solution, parallelization part including Mandelbrot Set computation and store temporary result of each thread. Serialization part including threads initialization, task assignment and merge all results of threads into one final result, the more threads, the more overhead in this part.

3.2 Evaluation of Static Partitioning

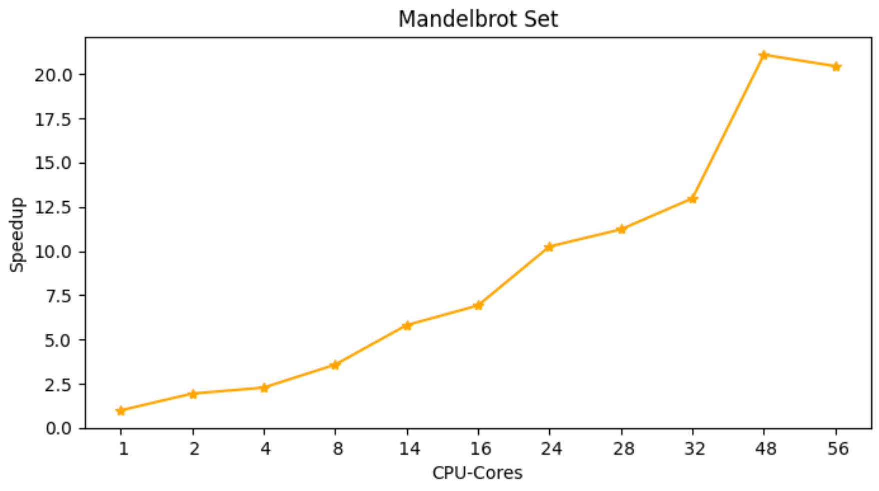

Figure 4. speedup x20 of optimization algorithm

Multithreads is used in this algorithm to achieve parallel execution on 56 cores machine. The figure above shows how the speedup changes with the number of cores by using **Static Partitioning** strategy. It can be concluded from the above graph that this strategy can achieve a speedup only to 20 while using 56 CPU cores.

According to Amdahl's Law,

$$S_{latency}(s) = \frac{1}{1-p+\frac{p}{s}}$$

It can be calculated from the evaluation of the strategy that in the promising strategy, the parallelization part accounts for 97.2% of the whole execution time in 28 cores, while it accounts for 96.9% of the whole execution time in 56 cores. The overhead of serialization parts increases as the number of threads grows.

In this solution, parallelization part including Mandelbrot Set computation and store temporary result of each thread. Serialization part including threads initialization, task assignment and merge all results of threads into one final result, the more threads, the more overhead in this part.

4. Comparison of the parallelization strategies

Both parallelization strategies are scalable.

For the load-balancing strategy. With increasing of data scale, each thread just needs to calculate more rows, but the workload of them is still almost same, no one would be the bottleneck and slow down the whole program. If data amount does not change but there are more threads, workload of each thread would decrease and the whole program will be faster.

For the static partitioning strategy. With increasing of data scale, every thread still is assigned only one task (one block), but the task size is also increased. However, workload of each thread is not same. If a thread is lucky enough to be assigned to a block which contains almost no points belong to Mandelbrot Set, it can finish its task within very short time. Conversely, if a block contains large number of points belonging to the Mandelbrot Set, corresponding thread has to spend more time to finish its job, these threads would greatly increase the running time of the entire program.

In summary, load-balancing is a better solution for concurrent problems. It is widely used in web service and software applications. It has satisfactory response speed and throughput.

Appendix

Measurement	Git Hash	Evaluation URL

Baseline	5f02e6f4	https://cds-lab.pages.se-gitlab.inf.tu-dresden.de/cds-s-2020/cds-website/logs/a0967e72cdb1108127661224e04c5f6bcac4d456d52090209b451d36429f223b/2020-06-25T09:16:26+02:00.log
Scalability Evaluation(x35)	a554212a	https://cds-lab.pages.se-gitlab.inf.tu-dresden.de/cds-s-2020/cds-website/logs/a0967e72cdb1108127661224e04c5f6bcac4d456d52090209b451d36429f223b/2020-06-14T07:51:11+02:00.log
Scalability Evaluation(x20)	a50070b6	https://cds-lab.pages.se-gitlab.inf.tu-dresden.de/cds-s-2020/cds-website/logs/a0967e72cdb1108127661224e04c5f6bcac4d456d52090209b451d36429f223b/2020-06-12T12:20:42+02:00.log